Kamila Szewczyk

KamilaLisp

THE COMPLETE REFERENCE

2021 - 2023

Abstract

KamilaLisp is a Lisp-inspired, dynamically typed programming language that borrows many ideas from Haskell, Standard ML, APL, Scheme and others. It is a functional programming language with a strong emphasis on array programming, and it is designed to be used in a wide range of applications. Its design stems from previous iterations, which featured more orthodox Lisp syntax, lazy evaluation and symbolic computation with an emphasis on mathematical programming.

Foreword

KamilaLisp as a language has its origins in its previous iterations, v0.1 and MalbolgeLISP. The first iteration of MalbolgeLISP (v1.0) was released in August 2020 and had very few of the features that distinguish KamilaLisp today. The second iteration of MalbolgeLISP (v1.1) was released in July 2021 and was the subject of much attention due to its unusual choice of implementation language. The final MalbolgeLISP (v1.2) was released in September 2021, sharing many ideas and features with KamilaLisp. Unfortunately, the effort required to implement it continued to grow. The codebase was becoming increasingly disorganised and difficult to maintain. In addition, the implementation of new complicated features was increasingly difficult due to the choice of implementation language. The first version of KamilaLisp, a MalbolgeLISP-inspired Lisp dialect, appeared in December 2021, featuring lazy evaluation, sophisticated numerical and symbolic operations, and many original design choices borrowed from MalbolgeLISP. Progress on the language has stalled due to significant performance problems with the Java Virtual Machine, the overhead of lazy evaluation, and some questionable decisions made at the core of the interpreter. After realising this rather early, an effort to create a C++ rewrite of the original code base began towards the end of 2021, but was quickly abandoned due to problems with garbage collection, efficient memory management, and C++ lacking certain qualities that would otherwise make it a good language in which to implement an interpreter for KamilaLisp. In August 2022 I started work on KamilaLisp v0.2, a Java rewrite of the original codebase that does not have some of the problematic features of the previous versions. This book, in the hope of being useful to the determined reader who wants to learn about KamilaLisp, describes my experiences in implementing a complex language runtime using Java, and also describes the language itself. It is assumed that the reader has an elementary knowledge of functional programming or of the APL programming language.

Contents

Glossary

In this book, the following assumptions are made:

- *Folding* a list `⍵` with a function `⍺⍺` is equivalent to putting `⍺⍺` between every element of `⍵`, also given the identity element of `⍺⍺`. One could assume binding to the left (`(⍵[1] ⍺⍺ ⍵[2]) ⍺⍺ ⍵[3]`), or to the right (`⍵[1] ⍺⍺ (⍵[2] ⍺⍺ ⍵[3])`), hence the names *fold-right* and *fold-left*. Unlike reduction, the result of *folding* an empty array is defined and equal to the identity element.

- *Scanning* a list `⍵` with a function `⍺⍺` is equivalent to mapping a fold over prefixes of a list. This definition of a scan has a $O(n^2)$ complexity regardless of the fold type. A more efficient, $O(n)$ definition of a *scan-left* exists (fold with intermediate steps).

- *Outer product* of function `⍺⍺` (`∘.⍺⍺`) and arrays `⍺` and `⍵` is the application of `⍺⍺` between every pair of elements from `⍺` and `⍵`. The resulting list has a depth given by `⍴,⍤⍴`.

- *Inner product* of functions `⍺⍺`, `⍵⍵` and lists `⍺` and `⍵` (`⍺⍺.⍵⍵`) is equivalent to folding the list obtained by putting `⍵⍵` between corresponding pairs of `⍺` and `⍵` with `⍺⍺`.

- A *higher-order function* is a function that takes another function as its argument.

- A *lambda expression* (in the context of KamilaLisp) is an anonymous function with static (lexical) scoping, meaning that it can see all the variables bound by its lexical ancestor, unlike to dynamic scoping implemented by older LISPs where a function can see all the variables bound by its caller(s). In both cases, the first bound variable found is used, allowing to shadow variables.

- A *dyad* is a two argument function.

- A *monad* is generally a single argument function. The book also uses it in a context of an abstract data constructor implementing the *bind* and *unit* functions.

- *Replicating* a list `⍵` according to list `⍺` is copying each element of `⍵` a given number of times (specified by the corresponding element of `⍺` or assumed to be zero). In KamilaLisp, `⍺` can be a scalar, in which case the contents of `⍵` are catenated to each other `⍺` times. If `⍵` is also a scalar, it's repeated `⍺` times to form a list.

- *Filtering* a list `⍵` with function `⍺⍺` is equivalent to mapping `⍺⍺` on every element of `⍵` (assuming `⍺⍺` returns a boolean value, i.e. `0` or `1`), and replicating `⍵` with the result of the mapping (`⍵/⍤⍺⍺¨⍵`).

- *Rotating* a list `⍵` by `⍺` elements is equivalent to dropping `⍺` elements from `⍵`, and joining them with `⍵` (dyadic `⍚`, or more illustratively, `(↓,↑)`).

- *Zipping* two arrays is the act of forming a list of pairs via the juxtaposition of corresponding elements from the given arrays. Alternatively, the pairs can be processed by a functor (*zipwith*).

- *Flattening* a list if the act of decreasing the list's depth by one level, enlisting all of the elements from the topmost sublists into a resultant list.

- *Mapping* a function `⍺⍺` over a list `⍵` is the act of processing each element of `⍵` with the function `⍺⍺` to produce a resultant list of equivalent length.

- *Partial application* is defined as fixing a number of arguments to a function, yielding an anonymous function of smaller arity.
- *Iteration* of a function over an argument is defined as evaluating the function over its result starting with the initial argument until a condition is satisfied. Sometimes the condition is numeric, or is defined as a dyadic function between the previous and current result (just `iterate`). A fixed point combinator could be implemented using partial application of deep equality (KamilaLisp: `bind iterate =`, APL: `⍣≡`). Iteration is faster and consumes less resources than recursion, assuming no tail call optimisation.
- *Function composition* (or `@`/`∘`, as KamilaLisp calls it) is taking an arbitrary amount of functions where each function operates on the results of the function before, except the last function, which gets all the arguments passed to the anonymous function yielded by composition.
- *Selfie* is a higher order function that duplicates the argument to the function it takes (i.e. `×⍨`, computes the square of its argument).
- *Commute* swaps the order of two arguments to a function. `⍺ f ⍵ ⟺ ⍵ f⍨ ⍺`.

Chapter 1

Initial considerations

This chapter discusses the basics of KamilaLisp. Throughout this book, the author will use the KamilaLisp interpreter to check and execute the declarations of a program one by one. The emphasis will be put on list processing and mathematical functions to form elementary understanding of the language.

1.1. Programs and variables

A KamilaLisp program is a sequence of declarations, which are executed in the order they are written. The first program presented in this book is shown below:

```
--> def a (+ (* 2 3) 2)
8
--> def b (* 5 a)
40
```

It consists of two declarations. The first declaration binds the identifier a to the integer 8, and the second declaration binds the identifier b to the integer 40, which follows frmo the intuitive understanding of the arithmetic operations. To the reader not accustomed with Lisp-like syntax, every element of the syntax tree that would otherwise be implicitly grouped by a language with usual arithmetical precedence rules is explicitly grouped by parentheses to form a list. The resulting values of variables can be determined as follows:

```
--> ?a
8
--> ?b
40
```

The question mark is a sign for the KamilaLisp interpreter not to evaluate the entire input as an expression, but rather, to query the value of what it refers to.

Every list besides the empty list (usually written as `()` or alternatively `nil`) has a *head* defined as the first element of it. When a Lisp program is evaluated, the *head* of the current list is assumed to be a callable value, while the rest of the list (also called the *tail*) is assumed to be a list of arguments.

Because the list of arguments to a function is evaluated, a perceptive reader could point out a potential issue - *How to introduce list literals in the code*? This question is indeed well-founded, since the list literal would be evaluated in order to pass it parameter to some callable object, hence the tail of the literal would be applied to its head, thus behaving undesirably and almost certainly raising an error. Every Lisp dialect addresses this issue in the same way using the quoting mechanism. Simply put, the quote prevents a list from being evaluated. To observe this behaviour, introduce two more functions called `car` and `cdr` to obtain respectively the *head* and *tail* of a list:

```
--> car '(1 2 3)
1
--> cdr '(1 2 3)
(2 3)
```

KamilaLisp follows scoping rules familiar from other programming languages, such as Scheme or C++ - *static scoping* (also called *lexical scoping*), where an attempt is initially made to resolve a variable in the current scope. If this approach fails, the variable is resolved in the scope of its lexical ancestors until either the interpreter finds an environment where the variable is bound, or raises an error regarding an unbound variable. Additionally, variables may be *shadowed*, as demonstrated below:

```
--> def my-list '(1 2 3)
(1 2 3)
--> car my-list
1
--> def my-list (cdr my-list)
(2 3)
--> car my-list
2
```

However, it is not possible to shadow pre-defined variables and functions:

```
--> def car 5
RuntimeException thrown in thread 1dbd16a6: def can not shadow or redefine built-in bindings.
    at entity def  2:1
    at def primitive function
```